An Extensible Implementation-Agnostic Parallel Programming Framework for C in ableC

Aaron Councilman

Submitted under the supervision of Eric Van Wyk to the University Honors Program at the University of Minnesota-Twin Cities in partial fulfillment of the requirements for the degree of Bachelor of Science, *summa cum laude* in Computer Science.

May 10, 2021

Acknowledgements

I would like to thank my advisor, Eric Van Wyk, for his advice and guidance throughout not only this project but most of my undergraduate career, and for having invited me to join his group in the first place. I would also like to thank my thesis readers George Karypis and Jon Weissman for their time and insight.

I would also like to thank the entire MELT group, especially Lucas Kramer and Travis Carlson, for their help and advice over the years.

Finally, I would like to thank my friends and family for supporting me, especially through the strange year that we have had.

Abstract

Modern processors are multicore and this trend is only likely to increase in the future. To truly exploit the power of modern computers, programs need to take advantage of multiple cores by exploiting parallelism. Writing parallel programs is difficult not only because of the inherent difficulties in ensuring correctness but also because many languages, especially low-level languages like C, lack good abstractions and rather rely on function calls. Because low-level imperative languages like C remain dominant in systems programming, and especially in high-performance applications, developing parallel programs in C is important, but its reliance on function calls results in boiler-plate heavy code. This work intends to reduce the need for boiler-plate by introducing higher-level syntax for parallelism, and it does so in such a manner so as to decouple the implementation of the parallelism from its semantics, allowing programmers to reason about the semantics of their program and separately tune the implementation to find the best performance possible. Furthermore, this work does so in an extensible manner, allowing new implementations of parallelism and synchronization to be developed independetly and allowing programmers to use any selection of these implementations that they wish. Finally, this system is flexible and allows new abstractions for parallel programming to be built on top of it and benefit from the varied implementations while also providing programmers higher-level abstractions. This system can also be used to combine different parallel programming implementations in manners that would be difficult without it, and does so while still providing reasonable runtime performance.

Contents

Chapter 1

Introduction

Modern computers, from smartphones to servers, use multicore processors allowing them to run multiple programs simultaneously. This is important to support the multiprocessing workloads most computers are subjected to: a user expects their music to keep playing while they play a game, and a multicore processor can easily handle this. Because recent improvements, and likely much of the future improvement, in processor performance is coming from increased core counts, rather than increased clock speeds, high-performance programs must be able to utilize multiple cores. To do this, programmers must write parallel programs in which they designate what pieces of the program can be run simultaneously. There are major challenges posed in this process, however, especially since many of the semantics of sequential code no longer hold in parallel. A classic example of this is that a variable initialized to 0 and then incremented twice in sequence will have the value 2 but if two parallel operations each increment the value it is possible that the result is 2 but it is also possible that the value is 1 because both operations read the value to be 0, incremented it to 1, and then wrote the value back into memory as a 1. Writing correct and efficient parallel code is very difficult because programmers need to not only figure out what can be parallelized but also how these pieces are synchronized to ensure that the result that is produced is correct.

The difficulty of writing parallel code can be exacerbated quite a bit by the language

being used; C is a relatively low-level programming language that makes writing very high performance code possible, unfortunately the language itself provides very little support for parallel programming, instead all support comes in the form of library functions, which require that programmers utilize function pointers and often significant amounts of boiler-plate code, which has no semantic meaning to it but is needed to pass data between threads. Synchronization is also error-prone since there is little error checking built into the functions provided, and what is provided is done through return values which are often simply ignored by programmers. Despite this, we are interested in writing parallel programs in C because it remains a common language in systems programming and allows very high-performance programs to be written.

This work intends to decrease the difficulties of writing multi-threaded programs in C by hiding the boiler-plate and error-checking code from the programmer by providing better syntax and abstractions for parallelism and synchronization. In addition, it facilitates optimizing such programs by making it possible to change between implementations of the parallelism and synchronization constructs used in the program without having to rewrite large portions of the program.

1.1 An N-Queens Server

Throughout, we want to consider an example of a problem that demonstrates a variety of different types of parallelism and synchronization, and for this we have devised an N-Queens server. This is a server dedicated to solving two forms of the N-Queens problem. The N-Queens problem is that of placing n queens onto an $n \times n$ chess board such that no two queens can capture each other (meaning that they are not on the same row, column, or diagonal). In particular, we are interested in a COUNT query where given some initial configuration of up to n queens on the board we want to count how many ways we can place the remaining queens onto the board to produce valid configurations. We are also interested

in a NEXT query where given some initial configuration of the n queens, we find the next configuration in order that is valid, where the order is defined by representing the board as a base n number. If we were to begin with all the queens in the first row and repeatedly execute NEXT queries, this would enumerate all solutions for the given n. Throughout, we will discuss the various types of parallelism and synchronization needed to realize this server and demonstrate how this work is useful in the construction of this server.

While the idea of this N-Queens server is somewhat contrived, it is worth noting that we will be presenting what we believe to be asymptotically optimal solutions to solve these problems, and we believe the implementations to be reasonably efficient. To date there is no known formula for the number of configurations for a given n, though it is of course known to be bounded above by $n!$. A related problem, the N-Queens Completion problem, which is the problem of taking some initial configuration and adding the additional queens to form a valid configuration, is known to be NP-Complete[9]. Because both the NEXT and COUNT queries described above search spaces no smaller than is needed to solve the N-Queens Completion problem, these problems must be NP-Hard as well. In addition, neither of these problems has trivially few solutions, because there are on the order of n^n inputs to the NEXT query and $n!$ non-trivial inputs to the COUNT query, there is a real need to calculate solutions on the fly as we cannot simply calculate all possible solutions ahead of time.

We will describe the server's design in more detail as we proceed, but we will provide a brief overview of its design here. Our design for the server will have five components: the input interface will receive queries from the network, read the request, and prepare it for processing; a request processor will then look at the query and determine how to produce the result which it will do by sending the query to the solver for NEXT queries or the solver for COUNT queries; finally, the query, now with its result, will be sent to the response handlers which send the query response over the network. As such, this program exhibits several types of parallelism: the I/O interfaces will rely significantly on blocking system calls to send and receive network requests, the query processing itself is a very quick operation that invokes

the solver and then sends the query on to have its result returned, and the query solvers themselves are both compute-bound operations. The COUNT query is susceptible to very fine-grain parallelism while, because of its determinism, the NEXT query is not susceptible to much parallelism of an individual query.

1.2 Roadmap

Chapter 2 provides background information on parallel programming (Section 2.1), parallel programming in C (Section 2.2), and extensible programming languages (Section 2.3). Our contributions begin in Chaper 3, where we present how parallelism is written in the ABLEC PARALLEL system. Chapter 4 then completes the description of this system by presenting details on how to achieve synchronization using this system. Chapter 5 demonstrates the power of composable parallel programming extensions by demonsrating a higher-level type-based system for synchronizing shared objects. Chapter 6 shows the performance of code written using this system, demonstrating that these higher-level language extensions provide reasonable performance. Finally, we discuss related work in Chapter 7 and future work in Chapter 8.

Chapter 2

Background

This chapter provides background useful for the work that follows. Section 2.1 discusses how parallel programs are written and various techniques that can be taken to achieve parallelism and synchronization in multi-threaded programs. Then, Section 2.2 describes various techniques for writing multi-threaded programs in C, and extended versions of C. Finally, Section 2.3 describes the ideas of extensible programming languages and discusses ableC, the extensible version of C used in this work.

2.1 Parallel Programming

This work focuses on multi-threaded parallelism, that is parallel programs where there are multiple threads of execution which can, generally, run independently. This parallelism is well suited to multi-core processors, and is a very general form of parallelism because the data and instructions that each thread is using can be different, making it a multiple-instruction multiple-data (MIMD) form of parallelism. Writing multi-threaded programs has two components: writing the parallelism that actually allows pieces to be run in parallel and writing the synchronization that ensures these pieces do not interfere with each other and eventually we can combine their results into the desired result. There are a variety of ways that parallelism and synchronization can be achieved, which we will discuss below.

It is worth noting that in modern computers there are a variety of other ways to achieve parallelism, for instance vector extensions, GPUs, and dedicated hardware accelerators. However, for simplicity, we limit our scope to multi-threaded parallelism.

2.1.1 Parallelism

OS Threads

The simplest way to achieve parallelism in a program is to ask the operating system to create a new thread of execution. Introducing parallelism in this manner makes the OS inherently involved: it allows the OS to schedule each thread independently and allows an individual thread to block on blocking system calls while other threads can continue to run. However, the creation of an OS thread has significant overhead and is limited by system imposed resource limits.

Operating system threads are particularly well suited to use-cases that involve blocking, such as many I/O system calls. OS threads can be used in compute-bound use-cases, but they must be used cautiously; if the number of OS threads exceeds the number of hardware supported threads performance will often degrade because the OS will continue to schedule all the threads and this will result in threads being interrupted more often and may cause threads to be moved onto different cores which can severely impact cache performance. As such, OS threads are best suited in situations where either the workload involves blocking or when the number of threads needed is known ahead of time to be less than the number of hardware threads. As such, OS threads are well suited for the I/O components of the N-Queens server, which rely a lot on blocking either for the I/O or waiting for responses to be ready to send.

It is worth noting that all of the techniques we will look at ultimately rely on OS threads, but the other parallelization techniques will interpose some user-space control systems between the programmer and the OS threads in an effort to provide certain benefits over direct use of OS threads.

Thread Pools

Thread pools are composed of a limited number of OS threads which are used to service a larger number of logical threads. The general idea is that each OS thread takes a piece of work, executes it to completion, and then picks another piece of work. In this way, and with relatively little overhead, we can resolve some of the problems that can arise with having too many compute bound OS threads running simultaneously. Thread pools are particularly well suited to problems with relatively coarse-grain parallelism where the problem can be split into a number of pieces that are each roughly the same size; and this split can be done immediately in the computation. In our N-Queens server, thread pools are well suited to handling the query processing and NEXT queries, since in both we are mostly interested in avoiding having an excessive number of OS threads and the performance issues that can arise from that.

The drawback of thread pools is that they are not well suited for blocking operations because the OS is only aware of the OS threads and so when a blocking system call is invoked the entire OS thread will block, even though there may be other work that the thread could be working on at that time. With systems calls the only widely available solution is to use non-blocking system calls instead, if possible. There have been proposed mechanisms to held address these issues [1], but these require support for the OS that is lacking from most operating systems. For thread synchronization, it is possible to avoid blocking by developing user-space techniques to block a logical thread by saving its state and allowing the OS thread to select another work item.

Work-Stealing

Work-stealing is a technique for parallelism that is particularly well suited to fine-grained parallelism where a problem can be broken into many pieces, especially when these pieces are revealed through recursion. While other implementations exist, we will consider the implementation of work-stealing in Cilk5 as it was a successful version, that became integrated

into both GCC and the Intel C compiler, and is described in some detail in the literature. In this approach the system has a number of OS threads, each with its own work deque (double-ended queue); when a thread reaches a point where it would split off a new thread it places a "closure" describing the function it is currently executing onto its deque and then begins executing the new thread's code. If other threads in the system are not working on anything at the time, they may come and steal the closure from the deque and continue working on that function. In this way, a work-stealer handles fine-grain parallelism without an explosion in the number of threads, and the system automatically load balances. The disadvantage of this approach is that there is significant overhead for managing the deques and closures needed by these systems. Because work-stealing systems are well suited to fine-grained parallelism in compute-bound problems, it is well suited for solving the COUNT queries in the N-Queens server, as these are search problems through large search spaces.

2.1.2 Synchronization

There are two approaches to synchronization that we will discuss: fork-join synchronization and mutual exclusion synchronization. The first of these is used for synchronizing tasks that are being executed in parallel while the later is used for synchronizing shared memory objects. While there are other approaches to synchronization, and there is even support in modern hardware for synchronization, we limit our discussion to these two approaches as they are the extent of synchronization provided in our system. We focus on these two approaches because they can be used to express all the synchronization needed in multi-threaded programs, though other implementations may be more efficient.

Fork-Join

Fork-join is a synchronization technique used to get values from parallel calculations, while this can be very useful it is a very limited form of synchronization since it does not handle "object synchronization" which are the issues that arise, for instance, when multiple threads

attempt to increment the same value. Ultimately, the idea of fork-join parallelism is that after forking computations to be performed in parallel, we eventually want to wait for some of those computations to complete so that the results of them are known. This operation, of waiting for those parallel computations to complete, is referred to as *joining* the forked computation.

Mutual Exclusion

The primary technique for synchronizing shared objects is using locks. The general idea of a lock is that only a single thread can hold the lock at one time, so for instance to increment a shared integer we would acquire the lock, increment the number and then release the lock. As long as all accesses to the shared object are synchronized using the same lock, there should be no issues. However, in exchange for this easy synchronization, locks are a very heavy approach since they produce sections of programs where only a single thread can run at a time. There are a variety of implementations we will consider:

Spinlocks The simplest form of a lock is a spinlock, so named because the threads waiting to acquire the lock spin, meaning that they continue to consume CPU time, generally by attempting some atomic operation that is used to determine which thread acquires the lock. Generally, spinlocks should be avoided because they waste processor time, especially if a thread holding the lock gets taken off the processor by the OS scheduler. Spinlocks are occasionally useful, or even necessary, in situations where the lock is held for a very short time but hardware synchronization will not work, likely because we may need to update several pieces of data atomically.

MCS Locks MCS Locks are another form of spinlock, though these are intended to improve performance in high contention settings. Multiple threads atomically modifying a single piece of memory in a tightly spinning loop causes cache thrashing, making standard spinlocks very inefficient. MCS locks resolve this by using a different memory location per

thread. This can be useful in high contention settings, though the other problems with spinlocks remain.

Mutexes Mutex locks, also called mutexes, are blocking locks; so rather than wasting processor time waiting on the lock, the waiting threads are put to sleep, generally by the OS. Because this involves system calls, these locks have more overhead than a spinlock, though the Linux implementation, commonly referred to as a "futex" is designed to spin for a little in user-space in case the lock is released quickly, before blocking. This is generally useful when the lock is held for a short time, so the small amount of spinning can often avoid the system call.

Locks, especially mutexes, are commonly associated with condition variables. A condition variable can be used to atomically release a lock and block the thread, pending a signal that awakens the thread. This can be used, for example, if we have a shared queue and a thread wants to remove an element from an empty queue, it can wait on a condition variable, and then when an element is added, we expect that to awaken the blocked thread, which can now remove and return the element.

Depending on how much contention there is for a particular lock and on how much work is done while the lock is held, there may be a variety of other implementations of interest. In addition, there are other lock-like constructs. For instance, read-write locks allow any number of threads to acquire the lock for reading but only allows a single thread at a time to acquire the lock to modify the object. This can be very useful when most operations just need to read the shared data, though there are issues that arise with how to handle writing and making sure updates eventually occur.

2.2 Parallel Programming in C

Because this work focuses on the C language, we will briefly discuss how the techniques described above can be achieved using C, via the POSIX thread API (`pthreads`)[10], as well as two extensions to C, OpenMP[13] and Cilk[8].

The only form of parallelism available in pure C is via the `pthreads` library, which allows for the creation of OS threads. This is done through the `pthread_create` function. This API call is used to create a thread which will invoke a specified function, which must take a void pointer as its one argument and return a void pointer as its result, with a specified argument. Because this interface only permits this one type of function, most uses of it require creating a `struct` to hold the various arguments transmitted to the function and placing these into it before invoking the API call. In addition, because this utilizes void pointers, it is unsafe because the compiler itself provides almost no type-checking. This API, though, is well suited for fork-join synchronization, as the `pthread_create` results in a value of type `pthread_t` which can then be used with `pthread_join` to join that thread, and even retrieve the return value if desired. In addition, the POSIX API provides spinlocks, mutex locks, and condition variables through specific types and associated functions for using them.

OpenMP introduces parallelism through `#pragma` directives written into the code, specifically it is used either to designate that a section should be executed by multiple threads or that the iterations of a for-loop should be divided over a number of threads. The implementation of both of these constructs is simply to create a specified number of OS threads. Because both of these constructs rely on adding annotations to the code, there is less boiler-plate than using the `pthreads` API, but it is more difficult in OpenMP to create threads which are executing different pieces of code. OpenMP also supports a thread-pool like approach to parallelism using tasks, however this is not explicitly a thread pool and the annotations needed are not necessarily intuitive making it somewhat difficult to understand the code. Examples of this technique can be found from van der Pas[17]. Synchronization in OpenMP is less flexible than it is using `pthreads`, but it is generally implicit, meaning that program-

mers do not have to consider it directly. It also provides pragmas for making certain sections atomic, but again this is less fine-grained than explicitly using locks.

Finally, Cilk, specifically Cilk5, is really just a work-stealing system. It uses a "two-clone" approach where each work-stealing function is translated into two versions, a fast clone used if the function is never stolen and so no synchronization is needed and a slow clone used when synchronization is needed because the closure was stolen. Because work-stealing is generally used only in compute-bound problems, Cilk has no facilities for mutual exclusion, though it does still have fork-join synchronization since this is needed to synchronize the spawned tasks. This synchronization is achieved by just writing `sync;` which will synchronize all previously spawned work in this function, so while it is simple it is also not very flexible. The actual implementation of Cilk5, and the syntax used, are described in more detail by Frigo et. al.[8].

2.3 Extensible Programming Languages

The C language has not changed very much since its inception, which is part of why it lacks features for parallel programming. A language like C does not change much because it is difficult to change an established language, and adding more features eventually creates a very cluttered and difficult to use language. Because parallelism is so important for modern programs "extensions" to the language, like OpenMP and Cilk are developed. However, these are difficult to implement, for instance the original implementation of Cilk5 used its own type-checker despite not changing anything about the type system.

The purpose of an extensible programming language is that new features and syntaxes can be added as small and independent "extensions" that only have to implement the new features and then plug this into a larger language. A programmer can then pick the extensions they want so as to include the features they want to use, and this may be a different set of extensions for different projects. In a good extensible language, the extensions themselves

can also be extended, meaning that extension developers can introduce new, often higher-level features by building off of other extensions, rather than having to build from the base language.

ableC is an extensible C11 pre-processor [11], which translates from an "extended C" language into standard C code. It operates by reading in the extended C code; performing static checks over the code, including regular C type checking; and if there are no errors it produces regular C code as the translation. Language extensions in ableC can introduce new syntax and specify arbitrary translations for this, and can include arbitrary static analysis as well. Extensions can also introduce new types which can overload the standard operators in C. ableC itself, and all language extensions to it, are written in Silver, which is an attribute grammar that is well suited to creating extensible languages. A particularly nice feature of Silver and ableC is that Silver provides analyses which, if extensions satisfy, guarantee that those extensions can be composed with other without any conflicts during compilation. This does impose some limitations, though these are primarily to do with the syntax to ensure that a parser can be generated for any composition of languages.

While Silver can guarantee that the language extensions can be composed however a user wants, it is also worth considering difficulties that can arise from runtime systems. For instance, there is a Cilk extension for ableC, however the Cilk5 runtime makes assumptions about its use that makes it very difficult for other parallelization to be used in tandem. We have previously explored some of these issues and developed some techniques to allow parallel systems to run as if they are running in isolation [6].

Chapter 3

Achieving Parallelism

In Chapter 1 we introduced the N-Queens server and described its general operation, in this chapter and the next two, we will present the code for this server, and demonstrate how our ableC parallel language extensions help in the development of parallel programs.

3.1 Parallelism in the N-Queens Server

To begin, we will consider the various workloads present in the N-Queens server and the types of parallelism that are appropriate for them. First, we receive requests via the network which is most easily achieved using blocking system calls; as such, this portion of the program is not compute bound, and since a lot of time will be spent blocking, we may be able to benefit from having a large number of these threads to ensure that the server remains responsive. Next, we consider processing the two query types; both of these will be compute bound, but they differ in how much they can be parallelized. A count query exhibits large amounts of parallelism since we must explore the entire search tree and we can easily do this in parallel. However, a next query does not parallelize as easily since it only needs to explore as much of the search space as is needed to find the next solution. Because they are both compute bound, we still want to limit the number of these queries run at a time, but we notice that count queries may benefit from work-stealing parallelism while parallelizing an individual

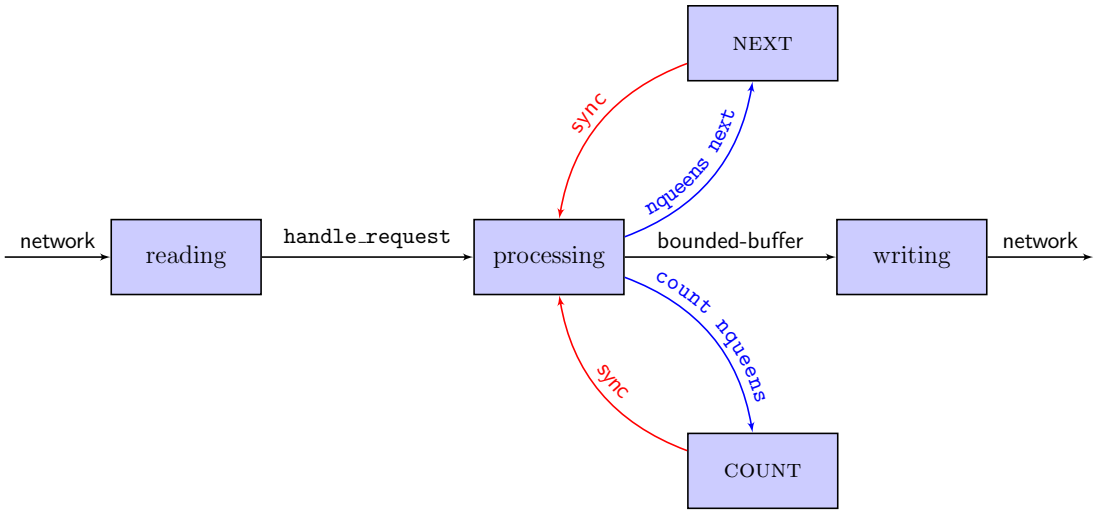

Figure 3.1: The N-Queens Server Design and Data Flow

NEXT query is unlikely to provide significant benefit. Finally, we must also send replies from the server, which again relies on blocking system calls.

Considering the behaviors of the different parts of the server, we settle on having five set of threads, each handling a different portion of the program. The design of this system, and the flow of requests through it, is shown in Figure 3.1. Requests come into the system from the network, shown by the incoming arrow on the left-hand side of the figure, to a set of reading threads, dedicated to reading network requests; these threads will block in I/O functions and since they only wake-up when they receive a request, we can have many of these threads operating concurrently. These reading threads then send the request to a processing thread pool by asking for a `handle_request` function to be executed in parallel in that thread pool. This function determines what type of request it is, sends the request out to compute the result (shown by the blue arrows in the diagram) and then waits for the result to be computed (shown by the red arrows). Once the result is computed, it then prepares the response to the query. While this work is relatively simple, we separate it from the reading threads so as to keep the server as responsive as possible and separate it from the query solvers to keep them dedicated to their operations. The processing task sends each request either to the COUNT work-stealer or the NEXT thread pool based on the query type, and

waits for the result from them. Finally, once the response has been prepared, the processing sends it along to the writing threads which will then send the response over the network. Sending requests from the processing to the writing threads is done using a bounded-buffer, a request is placed on it by `handle_request` and then pulled off the bounded-buffer by a writing thread. The writing is a separate set of threads since they utilize blocking system calls, both for managing the bounded-buffer and sending responses, and as a result we want to keep these threads separate from the thread pools handling compute-bound computations.

3.2 Parallelism with ABLEC PARALLEL

We will now look at how at how we write parallel code using the ABLEC PARALLEL language extensions. To do this, there are two components we must consider: the implementation-agnostic specification of parallelism and the specification of implementation details. For the first of these, we borrow significant inspiration from parallelism in Cilk and OpenMP; we provide abstractions for task spawning and parallel for-loops. While neither of these is a new abstraction for parallelism, because they are commonly used and so there is no question that the opportunity for parallelism supposed by these abstractions is sufficient to write useful parallel programs. For the second aspect, specifying the implementation, we write annotations on this code.

3.2.1 Specifying Parallelism

The first abstraction we provide is task spawning, with syntax inspired by Cilk. We write task spawning as `spawn` *expr*`;` *annotations*, specifying that the expression *expr* may be executed in parallel. The *annotations* are used to specify various implementation details of the spawn, and will be discussed in more detail in the next section. Even though we use an expression, the spawn does not produce a value, and any value produced by the expression is ignored, we simply use an expression because we expect most uses to be a function call (for

example `spawn foo(bar);`) or an assignment of a function call to a variable (for instance `spawn x = foo(bar);`). As a result, the task used with the spawn is generally executed for its side-effects, whether these are produced by a function call or modification of variables. We provide a few techniques for synchronization, which will be discussed in Chapter 4, which can be used to wait for the task to complete.

The other abstraction we provide for parallelization, inspired by OpenMP, is a parallel for-loop. A parallel for-loop allows the iterations of a for-loop to be divided into several blocks which are executed in parallel. This is generally used with for-loops where each iteration is independent; for instance applying a function to each element of an array. A parallel for-loop is written by writing `parallel` before the header of the for-loop. For example, to apply the function f to each of the n elements of an array arr we write the following:

```
parallel for (int i = 0; i < n; i++)
  annotations
{
  f(arr[i]);
}
```

Note that the *annotations* are used to specify various implementation details of the parallel for-loop, and will be discussed in more detail in the next section.

The body of the for-loop can contain arbitrary code, except that modification of the loop-variable is not permitted. The header of the loop, however, is more restricted: it must define a new loop-variable of an integer type, and the loop must be normalizable so it can be expressed in the form `for (int v = 0; v < l; v++)` where the original loop variable can then be calculated from `v`. This restriction is relatively minimal since most reasonable for-loops can be normalized in this manner. Like with a spawn, the for-loop does not produce any value, and so the iterations are generally executed for their side-effects. We will also discuss the synchronization of these loops later.

3.2.2 Specifying Implementation

The specification of various implementation details is achieved using annotations. Annotations are specified as a keyword followed by an expression and terminated by a semi-colon, and these annotations are placed directly after the spawn or parallel for-loop that they specify the implementation of. While ideally none of the annotations would carry any semantic meaning, some of them do, and so the annotations can be broken up into three categories: implementation, synchronization, and sharing.

Implementation Annotations

The implementation annotations provide no semantic meaning, and so these values should be able to be changed without any impact on the result of the program, only on the runtime performance. There are two implementation annotations, the simpler one is `num_threads` t`;` which is used on a parallel for-loop to specify the iterations should be divided into t blocks. This annotation is required for every parallel for-loop.

The other implementation annotation actually specifies the back-end that should be used to achieve the parallelism, so it is used to select whether the program should create new OS threads or send the work items to a thread pool, and in the later case determines which thread pool it is sent to. This annotation takes the form `by` *sys*`;` where *sys* specifies the system that should be used to parallelize the work. The system is specified using a *parallel interface*, which is discussed in Section 3.2.3. Because this annotation specifies the implementation, every spawn or parallel for-loop must have this annotation.

Synchronization Annotations

The annotations that are used with synchronization will be discussed in more detail in Chapter 4. These annotations take the forms `in` *group*`;` and `as` *thread*`;` and ultimately are used to provide names to tasks that can later be used to synchronize them.

Sharing Annotations

Based loosely on OpenMP, the last type of annotation specifies how variables should be shared between threads. For each variable used within the spawn or parallel for-loop we must specify whether that variable is a global variable, using the `global` *var*`;` annotation; whether the variable should be passed by reference to the parallel task, using the `public` *var*`;` annotation; or whether the variable should be passed by value to the parallel task, using the `private` *var*`;` annotation. Public variables allow a task to modify local variables, for instance to update the local variable `x` from a spawn, we would write `spawn x = foo(bar); public x;`. We require exactly one of these annotations for each variable used within the spawn or parallel for-loop. The sharing annotations can have a significant impact on the semantics of a problem. As a simple example, consider having `spawn x = foo(bar);` if `x` is made public then the local variable `x` is updated but if `x` is private then the local variable `x` is unchanged, also if it is set to global than it could modify a global variable that would otherwise be shadowed by a local variable.

3.2.3 Parallel Interfaces

The key to specifying parallelism in ableC parallel, while still keeping the system implementation agnostic, are *parallel interfaces*. For each parallelization system a programmer wants to use, they will create and initialize a parallel interface, and then use it on the spawn and parallel for-loops that should used that particular implementation. As a result, a parallel interface represents both an implementation of parallelism as well as possibly a particular runtime system, for instance in the N-Queens Server there will be multiple thread pools, each of which will have its own parallel interface.

To create a parallel interface, we declare a variable as *sys* `parallel threads;`. Here, *sys* is a type qualifier which specifies what implementation to use; for example we might have that `thrdpool parallel threads;` creates an interface to a thread pool, while `posix parallel threads;` creates an interface to POSIX thread creation. This qualifier is neces-

sary since the base ableC parallel system does not have any implementations, and rather extensions provide these. After declaring the parallel interface, it must be initialized using `threads = new` *sys* `parallel(`*args*`);`. Again, we require that the qualifier specifying the implementation be provided. The arguments to this may vary based on what the implementation is, for a thread pool interface the arguments likely include the number of threads to create for the thread pool, since this initialization process must create these threads.

Then, as mentioned above, the parallel interface is put to use with the `by` *interface*`;` annotation, providing a parallel interface to this to specify the implementation of the spawn or parallel for-loop.

Finally, when an interface is no longer needed, it should be destroyed to allow any allocated memory or threads to be de-allocated, and this is done by `delete` *interface*`;`.

3.3 The N-Queens Server

Having discussed the design of the N-Queens Server and how we specify parallelism in the ableC parallel system, we are now ready to take a look at how we use this parallelization system to implement the server. The first piece we will take a look at provides setup for the server, it creates the components shown in Figure 3.1, and we show some of the details of how the reading threads work and send requests on to the processing system. Throughout, code highlighted in blue shows new constructs introduced by the parallel system and orange shows annotation keywords, including the type qualifiers used on parallel interfaces. We can see this code in Figure 3.2. Note that lines 37, 47, 15, and 16 involve synchronization which will be discussed in Chapter 4.

Lines 1 through 5 declare the parallel interfaces we use. The first is the POSIX interface which is used for creating the reading threads, the next two are thread pools, one for the processing and the other for next queries, the fourth is the work-stealing system used for solving count queries, and finally there is another POSIX interface for the writing threads.

```
1  posix parallel reading;
2  thrdpool parallel processing;
3  thrdpool parallel next;
4  cilk parallel count;
5  posix parallel writing;
6
7  int main() {
8    // Setup elided
9    reading = new posix parallel();
10   processing = new thrdpool parallel(N_PROCESS);
11   next = new thrdpool parallel(N_NEXT);
12   count = new cilk parallel(N_COUNT);
13   writing = new posix parallel();
14
15   posix group readers; readers = new posix group();
16   posix group writers; writers = new posix group();
17
18   parallel for (int i = 0; i < N_WRITERS; i++)
19     by writing; in writers; num_threads N_WRITERS; global write_thread;
20   {
21     write_thread();
22   }
23   parallel for (int i = 0; i < N_READERS; i++)
24     by reading; in readers; num_threads N_READERS; global read_thread;
25   {
26     read_thread();
27   }
28
29   // Synchronization details elided
30   delete reading; delete processing; delete next; delete count;
31     delete writing;
32   // Clean-up elided
33 }
34
35 void read_thread() {
36   int res; struct request* req;
37   posix group grp; grp = new posix group();
38
39   while (1) {
40     req = accept_request(sockfd);
41     res = process_request(req);
42     // shutdown and error handling elided
43     spawn handle_request(req);
44           by processing; in grp; private req; global handle_request;
45   }
46
47   sync grp; delete grp;
48 }
```

Figure 3.2: Basic example of creating parallel objects and using spawn and parallel for

In main, line 9 through 13 provide the initialization of the parallel interfaces. For the thread pools and work-stealer, this initialization takes the number of threads to create as an argument, while the POSIX interfaces takes no arguments. We then have two parallel for-loops on lines 18 and 23 which create the writing and reading threads, respectively. These threads just invoke the read or write function. Since both parallel for-loops are given the same number of threads as there are iterations and the implementations use the POSIX interfaces, this creates one `pthread` for each reading and writing thread.

In `read_thread`, the function on line 35 we can see some setup, related to synchronization, and then the main loop. We then accept a request from the network, and read it and perform some basic error handling with the `process_request` function. Then, on line 43, the request is handled. We do this by sending the request to the processing pool, which is specified using the `by` annotation on line 44. In addition, we can see annotations specifying that `req` is passed by value to that task and that `handle_request` is a global function.

3.4 Design in Silver

The parallel system is split into multiple extensions, the core ableC parallel extension and other extensions which build on top of it. This core extension provides the syntax and some basic error checking for parallel interfaces, spawn statements, and parallel for-loops, while further extensions provide the implementations of parallel interfaces. This is achieved by the use of a type qualifier which is introduced by an extension wishing to provide a new parallelization implementation. Associated to this qualifier, the extension must provide an implementation of the `ParallelSystem` nonterminal. This nonterminal is defined in the ableC parallel extension and has four main components: a specification for `new` and `delete` and a specification for `spawn` and parallel for-loops. The later two of these, are of most interest since they are the most elaborate. An extension must provide two functions, *fSpawn* and *fFor* which take the annotations and expression or for-loop, respectively, and

are expected to translate the spawn or parallel for-loop into appropriate C code. These functions are invoked from the ableC parallel extension by finding the `ParallelSystem` associated with the parallel interface provided to the `by` annotation.

The *fSpawn* and *fFor* functions are expected to produce the behaviors described above, both in regards to the behaviors of the abstractions and the annotations. In particular, these functions are responsible for handling the sharing and synchronization annotations. The handling of synchronization will be discussed in more detail in Chapter 4, where the specification of threads and groups will be described. For variable sharing, the implementation is responsible for determining how to pass public and private variables to the new thread, it must ensure that it passes public variables by reference and private variables by value. In addition, for parallel for-loops it must evaluate the loop bound a single time and from this determine how to split the loop over the specified number of threads.

3.5 Library and Runtime

The ableC parallel extension also provides some library functions needed for its implementation or that may be useful for other extensions. The most interesting of these features is Thread Control Blocks (TCBs) which are used for thread identification. It is, in general, expected that each logical thread be associated with a unique TCB (in fact, these are used to identify whether a thread is holding a lock or not). A pointer to this TCB is stored in thread-local memory (using C's `_Thread_local` storage class) and in addition to providing thread identity it also provides information about the parallel system the thread is running in, including any specific information associated with that system, additionally thread-specific information can be stored in the TCB at the parallelization system's discretion.

3.6 Provided Implementations

As demonstration of how to develop parallel implementations, we have developed three extensions to the core system, each providing a different implementation for parallelization. The first simply creates a new OS thread (using the POSIX API) for each spawn and each thread executing a parallel for-loop; the second creates a thread pool that work is sent to; and the final one provides a work-stealing system that work can be sent to.

3.6.1 POSIX-Based Parallelization

The POSIX implementation is very straightforward: a spawn creates a new thread which executes the given statement and a parallel for-loop creates the specified number of threads, each of which executes some iterations of the for-loop. The actual translation to C code is relatively simple, for ease we will consider a spawn, though the implementation for a parallel for-loop is similar. To translate a spawn to C code, we create a new `struct` which contains the variables needed by the expression, either as values if they are private or as pointers if they are public, as well as some information for each thread or group that is used for synchronizing this thread. We then produce a function which takes this `struct`, extracts variables from it, executes code needed for the threads and groups, and performs the actual expression that was spawned. Another interesting point, though, is that the actual expression must be modified slightly so that any public variables are changed from simply using the variable to instead dereferencing a pointer to the variable; the code that performs this transformation is actually provided by the ABLEC PARALLEL extension as this functionality is needed in many implementations. Finally, where the spawn occurs, we populate the `struct` and create a new thread using `pthread_create`. Because we do not make use of thread joining as defined in the POSIX API, these threads are set to a "detached" mode that allows their resources to be deallocated once the thread has completed. This is necessary to avoid memory leaks.

3.6.2 Thread Pool

The purpose of the thread pool implementation is to create a specified number of threads at initialization and then when we spawn threads they are placed onto the thread pool's work queue. The threads in the pool wait until work is placed onto the queue, and then take work items off the queue. These items are then executed. Like with the POSIX implementation the body of the spawn or the for-loop to be parallelized is lifted into a new function and a new `struct` is declared to hold the needed arguments to this function. The actual implementation of the `spawn` or parallel for-loop is also similar to that of the POSIX implementation: first we fill the `struct` and then we create the tread. However, in this case to create the thread we simply add it to the thread pool's work queue.

There is one interesting detail of the implementation of this system that is worth further discussion. To allow us to block a logical thread rather than an OS thread each work item is allocated to its own stack and when a logical thread wants to block it places itself onto a queue based on how it will be unblocked, and then jumps back into the thread pool scheduler using the C `longjmp` function. This allows, as we will see in the next chapter, to design locks that block logical threads in the thread pool, rather than blocking the OS threads that form the thread pool.

3.6.3 Work-Stealer

Finally, we have the work-stealing system. Like with a thread pool, we initialize a work-stealing system to have some number of OS threads and then send work to the system by placing a work item onto the deque of one of the threads; when this happens, some thread will eventually pick up the work and begin processing it. The actual implementation of the work-stealer, and the code that is generated for work-stealing functions, is strongly inspired by Cilk5, but is a new implementation. The primary reason for this is that the Cilk5 runtime system is intended to be run on its own, and in fact the runtime provides a `main` function that then calls the user written main function. Because of this, the runtime makes certain

assumptions that do not work with the flexibility this system allows. In addition, the code generation is also new because there is little published detail on the translation process used by Cilk5. Our implementation does borrow some from a previously developed Cilk extension for ABLEC[11], developed based on the cilk2C translator built for Cilk5.

Each thread in the work-stealer has its own deque and when the thread is not working on something, it will first attempt to remove a work item from its deque and if its deque is empty it will then begin attempting to steal from other threads deques. It does this in a circular manner, beginning with the next thread's deque, eventually looping back around to the beginning, and eventually its own deque. Currently, it will continue doing this until it finds work on some deque. It is actually important to note that a thread will try to steal from its own deque because we had previously encountered an issue with the Cilk5 runtime where deadlocks could occur because threads that were attempting to steal work would never check their own deques for work.

Functions that can be invoked in the work-stealing system must be declared to allow this, since these work-stealing functions require non-trivial code generation. To do this, a function is declared by prefixing the `cilk_func` keyword to the function signature or declaration. Within the function body, any valid ABLEC code is allowed, but in addition we can use `spawn` without any annotations to spawn other work-stealing tasks in this work-stealing system. This use of `spawn` is more restrictive as it must either be a function call, or an assignment where the right-hand side is a function call, to a work-stealing function. Then, to synchronize all of the previously spawned work-stealing tasks, we use `sync;`. This syntax is very similar to that used by the Cilk5 system.

In the N-Queens Server we make use of the work-stealer for the COUNT problem. To answer this query we construct a board based on the initial state provided by the query, and then proceed row-by-row through the board, testing each column to determine whether a queen can be placed there. If a queen can be placed there, we count the number of configurations that have a queen placed there, this is done recursively, and can be done in

```
1  cilk_func int count_helper(int n, chess_board* board, int r) {
2    while (r < n && board->order[r] != '\0') r++;
3    if (r == n) return 1;
4
5    int* cnts = calloc(n, sizeof(int));
6    for (int c = 0; c < n; c++) {
7      if (nqueens_filled(board, r, c)) continue;
8
9      chess_board* new_board = copy_board(board);
10     nqueens_set(new_board, r, c);
11
12     spawn cnts[c] = count_helper(n, new_board, r+1);
13   }
14   sync;
15
16   return sum_array(cnts, n);
17 }
18
19 cilk_func int count_nqueens(int n, char* init) {
20   chess_board* board = construct_board(n, init);
21   if (board == NULL) return 0;
22
23   int res;
24   spawn res = count_helper(n, board, 0);
25   sync;
26
27   return res;
28 }
```

Figure 3.3: Example of how to create work-stealing functions

parallel. We can see code for this in Figure 3.3, which demonstrates the use of work-stealing functions.

Looking through the code we can see the `cilk_func` keyword we described, as well as the use of `spawn` without annotations to specify a work-stealing spawn.

The compilation technique for work-stealing functions produces two copies: a "fast" and a "slow" clone. The fast clone has no synchronization and is used when the work has not been stolen and the slow clone includes synchronization and is used when the fast clone cannot be. The behaviors of these and the implementation of the fast clone are well described in the original Cilk5 paper [8]. The slow clone is more complicated since it must include synchronization between parallel components. The actual implementation of this in Cilk5 is not described in the literature; our approach uses an atomic integer as the "join counter"

and we do not proceed from a `sync;` until all previously spawned work has completed.

The implementation of `spawn` (in non work-stealing functions) is also somewhat interesting. Like with the other systems, we create a new function but in this case it acts like a slow clone which spawns the desired function and then waits for the result to be returned. It then takes that result and writes it back to an appropriate variable if needed. There is, of course, additional code added to update the threads and groups the spawn is added to. Then, to actually perform the spawn, we create a closure for the function including the needed arguments and add the closure onto some thread's deque.

One limitation of our current work-stealer is that it only supports spawn operations, not parallel for-loops. The reason for this is that what is allowed in the body of a parallel for-loop would have to be restricted to function calls to work-stealing functions or assignments where the right-hand side is such a function call. While this may be useful, it seemed that a normal for-loop containing a spawn could achieve this with nearly the same performance. In addition, because of the large number of work-stealing threads that can be created, we currently do not create TCBs for each thread, meaning that work-stealing functions cannot safely be used with synchronization. We suspect this is not a major limitation, since work-stealing systems are only really useful for compute bound operations.

Chapter 4

Achieving Synchronization

In this chapter, we will discuss the synchronization techniques provided in the ableC parallel system. As described in the background, there are two main techniques: work-join synchronization used to wait on results from parallel tasks and mutual exclusion used for synchronizing global memory.

4.1 Synchronization in the N-Queens Server

In Section 3.1 we described the parallelism that our N-Queens server will use, and so we now turn our attention to the synchronization used between the five components described. The first interaction is between the reading threads and the processing pool, each request is sent from the thread that read it to be processed asynchronously, and the reading thread should then resume reading other requests that arrive over the network. As such, and because the processing does not produce any value needed by the reader thread, we will not synchronize the queries individually. Instead, we will use synchronization when the reader thread receives a shutdown signal to ensure that all of the requests that thread handled have completed, as this means that once all the reading threads have finished all queries that were received have finished processing as well.

The processing pool is where the interesting synchronization occurs. Since both query

types are handled in different pools, the searching thread pool and the counting work-stealer, the processing will spawn work into the appropriate one of these and then join that task so as to wait for the completion of the computation. Once it has this, it will place the, now solved, request onto a queue that is used to send messages to the writing threads, which will pull the requests off this queue and then transmit the response.

4.2 Synchronization with ableC parallel

Now, we will discuss how to write synchronization using the ableC parallel language extensions. For this, we provide two distinct synchronization techniques: fork-join synchronization and mutual exclusion achieved using locks and condition variables. Again, the ableC parallel extension itself is implementation agnostic, and so the implementations of these constructs are achieved using annotations, specifically type qualifiers which are provided by extensions to the core system.

4.2.1 Fork-Join

The first abstractions we provide for synchronization are fork-join abstractions, allowing the programmer to demand that certain parallel tasks be completed before the program continues. This is achieved using `sync` *vals*`;`, where *vals* is a comma-separated list of *fork-join values*. These values allow the programmer to designate tasks that the completion of should be waited for. These values come in two types: a `thread` which represents just a single task or a `group` which represents any number of tasks. Synchronization using a `thread` waits for the associated task to complete while synchronization using a `group` waits for all associated tasks to complete. In keeping with the implementation agnostic nature of this system, the actual implementation of synchronization using either of these is left to extensions. Like with parallel interfaces, the implementation to use is specified using a type qualifier, so the declaration of a fork-join value has the form *qual* `thread thd;` or *qual*

`group grp;`. While a group is strictly more general than a thread, we include both because the thread is a nice abstraction in that it allows us to know (and using runtime checks guarantee) that only a single task is associated with it; in addition, it seems reasonable that certain implementations may exist where the implementation of a thread and a group might diverge significantly. Both types of values are initialized using `new` and are destroyed using `delete`.

Having described how the fork-join values are used with to synchronize tasks, the remaining question is how it is used when we fork parallel tasks. This is achieved, as mentioned briefly earlier, using annotations on the spawn or parallel for-loop. This is done using the `in` *group*`;` and `as` *thread*`;` annotations, where the *group* is some `group` value and *thread* is some `thread` value. These signify that the task, or in the case of a parallel for-loop tasks, should be associated with that fork-join value. Since parallel for-loops are considered to create multiple tasks, the `as` annotation can only be used with a spawn.

4.2.2 Mutual Exclusion

The other abstraction for synchronization is mutual exclusion, useful for synchronizing shared memory. Like in the POSIX API, we provide locks and condition variables to achieve this. In addition to the normal semantics of locks and condition variables, we also include error checking which verifies that a lock is released by the holding thread, there is not an attempt to reacquired the lock by the holding thread, and that any operation on a condition variable is made while the associated lock is held. While the POSIX API includes many of these error checking operations, they simply return error codes, rather than producing error messages and program termination which our implementations provide; our solution is more robust since it completely prohibits these behaviors and is easier since it does not require writing error checking code for each lock or condition variable operation.

Locks and condition variables are declared using *qual* `lock` *lk*`;` and *qual* `condvar` *cv*`;`, respectively. Again, like with fork-join values, the locks and condition variables are imple-

mentation agnostic and so the implementation is specified using a qualifier. They are both initialized using `new` and destroyed using `delete`. In our provided implementations the initialization of a lock requires no arguments while the initialization of a condition variable takes a single argument, the lock to be associated with the condition variable. As such, we might initialize a lock and a condition variable as `lk = new` *qual* `lock();` and then `cv = new` *qual* `condvar(&lk);`. Note that, in all our implementations, the lock and condition variable must have the same implementation, in other words the qualifiers must be the same.

Locks are acquired using `acquire` *lock*`;` and released using `release` *lock*`;`. The semantics of this, of course, is that at most one thread holds a particular lock at a given time, with the additional error checking discussed above. None of our provided implementations support recursive locks, though other implementations could. With condition variables, there are the same three operations provided in the POSIX API: waiting on a condition variable is achieved by `wait` *condvar*`;`, signaling by `signal` *condvar*`;`, and broadcasts by `broadcast` *condvar*`;`. The big difference we can observe between these and the POSIX API is that the wait operation does not require the associated lock be provided, because this is expected to be provided at the initialization of the condition variable.

4.3 The N-Queens Server

We will now look at how we make use of synchronization within the N-Queens Server. As described earlier, the N-Queens Server will make use of fork-join synchronization by the reader threads to synchronize the requests it sends for processing, as well as during the processing of a request to synchronize the computation. The synchronization used by the reading threads was shown earlier in Figure 3.2. On line 37 of that code, we see the declaration and initialization of a group, with the `posix` qualifier designating the implementation. Then, on line 44, we can see a use of the `in` annotation to specify that the spawn should be added to the tasks associated with that spawn. Finally, line 47 shows the synchronization of this

```
1  void handle_request(struct request* req) {
2    int n = req->n, res;
3    blocking thread thd; thd = new blocking thread();
4
5    if (is_next_query(req)) {
6      spawn res = nqueens_next(n, req->data); by searching; as thd;
7              global nqueens_next; private n; private req; public res;
8    } else {
9      spawn res = count_nqueens(n, req->data); by counting; as thd;
10   }
11
12   sync thd; delete thd;
13   set_response(req, res);
14   buffer_put(to_deliver, req);
15 }
```

Figure 4.1: Example of using a thread object to achieve fork-join synchronization

group and then the destruction of the group. As described, what this achieves is ensuring that when a reader thread exists it waits for all of the requests it read to be completely processed. We can also see, one lines 19 and 24, the use of a group with a parallel for-loop from the main; while the details are not shown these are used during shutdown to wait for the readers and writers to exit so that the server can be shutdown cleanly.

The `handle_request` function, spawned by the reader threads, is very simple, it determines what type of query has been made and then sends the computation to the searching or counting system. Then, it waits for the computation to complete and sends the response to the writing threads. The code for this is shown in Figure 4.1. Similar to the reading threads, on line 3 we create the thread and initialize it. Here, the implementation is specified by the `blocking` qualifier. Then, we determine the type of the query and spawn the `nqueens_next` or `count_nqueens` functions into the searching thread pool or counting work-stealer, respectively. In both of these, since we are using a thread for synchronization, we use the `as` annotation, as can be seen on lines 6 and 9. We then synchronize and delete the thread on line 12 so that the code that follows is not executed until that computation is complete.

The remaining synchronization comes from the queue used to pass completed requests from the `handle_request` function to the writing threads. In particular, we implement a

bounded-buffer, which is a circular fixed-length array-based queue implementation. As such, there are two operations: putting an element into the buffer which should block the thread if the buffer is full, until the buffer has space, and removing an element from the buffer which should block the thread if the buffer is empty, until the buffer has an element. Therefore, the implementation of the bounded buffer will use a lock whenever an operation attempts to add or remove an element from the buffer. This lock is necessary since we must update several fields atomically: the number of elements in the buffer, the index of the first or last elements, and for an insertion the entry at the last element. Two condition variables are used, one that is waited on when removing an element from the buffer if it is empty, and is signalled when an element is added, and another that is waited on when trying to add an element to the buffer and it is full, and is signalled when an element is removed. The code for this can be seen in Figure 4.2. We can see the declarations of the lock and condition variable on lines 4 and 5, here both the lock and condition variable have the `blocking` qualifiers to specify their implementation. The initialization of the lock and condition variables, on lines 9 through 11 shows that the condition variables are initialized with reference to the associated lock. We can then see acquiring the lock on lines 19 and 36 before making any changes to the bounded buffer. Then, in both the get and put operations we check whether the buffer is empty or full, respectively, and if so wait on the condition variable, as we can see on lines 22 and 39. Once we get past that loop we know that the buffer either has elements or has space, respectively, we make the needed modifications to the buffer, and then signal the appropriate change (so removing an element from the buffer using get means that the buffer is no longer full, and adding an element means it is no longer empty) using the signals on lines 29 and 46. Finally, we release the lock on lines 30 and 47.

```
1  typedef struct {
2    int head, tail, n, len;
3    struct request** items;
4    blocking lock lk;
5    blocking condvar empty, full;
6  } bounded_buffer;
7
8  void initialize_buffer(bounded_buffer* buffer, int len) {
9    buffer->lk = new blocking lock();
10   buffer->empty = new blocking condvar(&(buffer->lk));
11   buffer->full = new blocking condvar(&(buffer->lk));
12
13   // Other Initialization Elided
14 }
15
16 struct request* buffer_get(bounded_buffer* buffer) {
17   struct request* res;
18
19   acquire buffer->lk;
20
21   while (buffer->n == 0) {
22     wait buffer->empty;
23   }
24
25   res = buffer->items[buffer->head];
26   buffer->head = (buffer->head + 1) % buffer->len;
27   bufer->n -= 1;
28
29   signal buffer->full;
30   release buffer->lk;
31
32   return res;
33 }
34
35 void buffer_put(bounded_buffer* buffer, struct request* elem) {
36   acquire buffer->lk;
37
38   while (buffer->n == buffer->len) {
39     wait buffer->full;
40   }
41
42   buffer->items[buffer->tail] = elem;
43   buffer->tail = (buffer->tail + 1) % buffer->len;
44   buffer->n += 1;
45
46   signal buffer->empty;
47   release buffer->lk;
48 }
```

Figure 4.2: A bounded-buffer using a lock and condition variables

4.4 Design in Silver

Like with parallelism, the syntax and handling of qualifiers for synchronization is handled by the ableC parallel extension itself while the actual implementations are provided as extensions. For this, fork-join and mutual exclusion synchronization are treated separately; while we provide the same two implementations of both, there may be situations where a very particular implementation of fork-join synchronization or locking is of interest and so there should be no need to implement the other form as well. Like with parallelism, extensions provide an implementation of a nonterminal, `SyncSystem` for fork-join synchronization and `LockSystem` for mutual exclusion synchronization. The `LockSystem` is relatively simple: it must provide C types for the lock and condition variable, implementations of `new` and `delete` of both, and implementations of acquire and release for locks and wait, signal, and broadcast for condition variables.

The `SyncSystem` nonterminal has more components: it must provide C types for `thread`s and `group`s, handling of `new` and `delete` for both, and then a variety of functions for actually using them. Obviously, one of these components is an implementation of `sync` on the threads and groups, and the other component is an implementation of what the tasks need to do. This piece comes in three pieces: code that should be executed synchronously before the tasks is created, code that should be executed once the task's TCB has been created, and finally code that should be executed once the task is complete. Our implementations actually only use the first and last of these, but the middle is provided for if an implementation had need to track each thread's TCB. The first component is important because it allows a thread or group to be marked in such a way as to know that a thread has been associated with that object before that task actually starts running, which is important as otherwise there is a race condition between reaching the `sync` and the task updating the `thread` or `group`. These operations on the `thread` and `group` must be included in the generated code for a spawn or parallel for-loop, and this responsibility falls on the parallelization implementation.

4.5 Provided Implementations

To demonstrate how we can provide synchronization implementations, we have built two extensions to provide these implementations. One uses POSIX mutexes and condition variables and the other makes use of system-specific blocking functions so as to allow a thread pool (or other similar parallelization system) to block logical threads rather than OS threads.

4.5.1 POSIX-Based Synchronization

The POSIX implementation makes use of the mutex locks and condition variables provided by the POSIX API, though with the additional checks described earlier. The implementation of this is straightforward; a lock in this implementation is a `struct` which contains both a POSIX mutex and a pointer to the TCB of the thread which holds the mutex. Then, to acquire the lock we first check whether the TCB of the current thread matches that of the thread which holds the lock and if this is the case, we produce an error message. Otherwise, we simply acquire the lock using the POSIX API. Release, is similar though in that case we verify that the current thread is holding the lock. For a condition variable, the implementation is as a `struct` holding a POSIX condition variable and a pointer to the lock it is associated to, and we use that pointer to check that the thread operating on the condition variable is holding the associated lock.

Now, in this implementation threads and groups actually have the same type, a `struct` which holds a POSIX mutex and condition variable and an integer counting the number of threads that have started and not yet joined. Then, when a thread is being created associated with the thread or group, it acquires the lock and increases the thread count, and when it completes its work it acquires the lock and reduces the thread count. If it sets the thread count to 0, it then broadcasts on the condition variable to awaken any waiting threads. Therefore, to implement `sync`, we simply acquire the lock check whether the thread count is 0, and if not wait on the condition variable. There are additional checks in place to

ensure that a `thread` is not already in use, to ensure that we do not attempt to synchronize on an unused `thread`, and to allow `thread`s to be reused once it has been synchronized, but this is relatively simple and just requires tracking some additional state information.

4.5.2 Blocking

The "blocking" implementation, which makes use of system-defined `block` and `unblock` functions allows logical threads in a thread pool to be blocked by a lock, condition variable, or synchronization, rather than requiring the OS thread to block. Unsurprisingly, however, the implementation is significantly more complicated. As mentioned, this implementation relies on `block` and `unblock` functions defined by the parallelization systems. We will discuss these functions more in the next paragraph, but in essence they do what they say, when a thread calls its `block` function it will stop running until `unblock` is called with the thread's TCB as its argument. Therefore, we can actually implement a lock by simply keeping a queue of threads (using their TCBs) and if the lock is unheld, a thread simply acquire the lock and marks it as held, and otherwise the thread adds itself to the end of the queue and blocks. Then, when the lock is released, if there are waiting threads, the first one in the queue is awoken by calling `unblock`. To actually make this all work, we use a spinlock to ensure atomic accesses to the lock's data; additionally, when a thread is woken, it must still actually acquire the lock, it is not automatically given the lock.

The `block` and `unblock` functions work as described above, with the added wrinkle that they must work regardless of the calling order. Specifically, it is not guaranteed that `block` has even been called when the associated call to `unblock` is made, and these functions must operate so that given any interleaving of these two functions, the thread will return from `block` after, and only after, an associated `unblock` call has been made. Currently, we have versions of `block` and `unblock` for both the POSIX and thread pool parallelization systems but lack such an implementation for the work-stealing system due to technical difficulties involved in making such an operation work.

Chapter 5

Type-Based Synchronization

Locks and condition variables are commonly used to ensure exclusive access to shared objects, but doing this requires caution because it is easy to forget to acquire or release the lock, and it can be easy to forget when and how to generate signals and broadcasts on condition variables. The problem that we encounter using locks and condition variables in this way is that there is no static enforcement of the synchronization, and in fact the semantics of the condition variables are not defined in the language, at best they are described in comments that explain what a particular condition variable is used for. In this chapter we introduce another extension, building on top of what we have already described. This extension formalizes the synchronization semantics for shared objects by adding the synchronization semantics into the type of synchronized values and then using this to statically enforce the semantics and even automatically generate the code for dealing with condition variables. This allows the programmer to specify, in one place, the semantics of the object and then simply use the object without having to worry about writing the code to have appropriate semantics.

5.1 Synchronized Types

First, we will consider how we write these *synchronized types* in code and how we specify various aspects of synchronization semantics. Later, we will look at how values of these types

can actually be used.

5.1.1 Specifying the Type

Synchronized types are produced using a type constructor. To this constructor we specify the actual data to be contained in the value, and can specify the semantics of the condition variables associated with the value. The simplest form of a synchronized type can be written just as *qual* `synchronized<`*type*`>`. This will produce a type which behaves like the given type, except that it will require exclusive access to the object to read or modify the value. This, on its own, can be somewhat useful as it provides us static enforcement of appropriate locking, but we can also go further and specify condition variables that can be used with the value. Like with the locks and condition variables we described in the last chapter, the synchronized type is qualified to specify the implementation, and the implementations used with these are actually directly taken from the implementations defined for mutual exclusion synchronization, as described above.

Now, to create a synchronized type with condition variables, we use the syntax

qual `synchronized<`*type*`> =` {*conditions changes*}

This body has two pieces, *conditions* is a list of the conditions that the variable has and the *changes* describe what changes to the value result in changes to those conditions. Roughly speaking, the conditions are analogous to condition variables in the traditional approach. To define a condition, we write `condition` *name*`(`*expr*`);` where the expression *expr* is some expression that describes when the condition is true; this expression is often the same as (or the negation of) the condition that would be used for the loop surrounding a wait on the condition variable. For example, with the bounded-buffer definition we considered in the last chapter, we might define the empty and full conditions using the following code

```
condition empty(this.n == 0);
condition full(this.n == this.len);
```

In defining these conditions, we can use `this` to represent the instance of that synchronized type that the condition is being tested on.

Then, after defining the conditions, we describe what causes these conditions to change, this is equivalent to defining when condition variables are signaled or broadcast. The general form of this, in the code, comes as `when` *trigger* `then` *action*. There are a variety of options for both the triggers and actions we take, and we will discuss these next, but first as an example, with the bounded buffer conditions we might define their signals as follows

```
when (this.n) += 1 then signal not empty;
when (this.n) -= 1 then signal not full;
```

This is saying that when the number of elements is increased we know the buffer is not empty and when the number of elements is decreased we know it is not full. Now, turning our attention to the actions that can be used, the basic three are `signal`, `broadcast`, and `ignore`. The first of these have the same behaviors as their associated actions on a condition variable, waking a single thread or all waiting threads. The `ignore` actions is used to specify that a certain trigger is permitted but that it should generate no signals. This may be useful because we restrict what changes are allowed to a value if the value is specified as a trigger for some signal action. Concretely, if we define the empty condition for the bounded-buffer as shown above, then we might define triggers for when `this.n` is incremented or decremented. As a result, however, the system will now allow us to set `buf.n = 0` (where `buf` is a bounded buffer), since this assignment is an undefined action on `n`. We restrict actions like this so that the analysis and generation of signals can be done statically. We can also use `signal not` or `broadcast not` to designate that we should signal or broadcast that the condition is not true.

Finally, we consider the actual triggers of signals. These always have the form `(`*access*`)` *op val*. The access portion is some value derived from `this`, it can be the value `this` or field accesses, dereferencing, or array accesses. With array accesses, the index must be an underscore, such as `this.arr[_]`, which stands for an access to any element of the array. Next, we will consider the *op* and *val*. There are four options for *op* currently: `+=`, `-=`, `=`,

and `!=`. With either of the first two, *val* can either be a non-negative integer constant or an underscore. In the former case, the rule is triggered when the value is increased or decreased by the given constant value, and in the later case rule is triggered anytime we use the operator on that value. Note that with a trigger like `(this.n) += 1` writing something like `int a = 1; this.n += a;` is not permitted since we must able to determine the triggers at compile-time. Finally, with either the `=` or `!=` op, the *val* must be an integer constant; with the `=` operator the trigger is activated anytime we assign that value to the element and with `!=` the trigger is activated when we assign any other value to the element. As an example, considering a bounded-buffer and the empty and full conditions we described above, we would signal them based on updates to `this.n` and so we would describe the changes as follows:

5.1.2 Using a Synchronized Type

Next, having described how we define a synchronized type, we consider how we put it to use and how it guarantees that we are following the semantics defined in the type. The intention of the synchronized type is that is that it can be used almost as if it was just a value of its enclosed type. However, we are restricted some in what we can do because of the mutual exclusion guarantees that the type must provide; in fact we cannot access, either to read or modify, any synchronized value unless we have exclusive access to the value. When working with explicit synchronization we acquire exclusive access by acquiring some lock and then we release the lock when we no longer need it. Because this is a common pattern, but also something easy to mess up by forgetting either the acquire or release, for synchronized values this acquire and release is rolled up into one construct, a `holding` block. The idea is that we specify a synchronized value that we want exclusive access to, and then execute the body of the block once we have acquired the exclusive access, and release the exclusive access when the end of the body is reached. The holding block is written as

`holding (`*val*`) as` *name body*

where *val* is the synchronized value we want to acquire, *name* is a new name given to that synchronized value, and *body* is the statement to execute with the exclusive access.

Even within a `holding` block there are still limits on how we are allowed to modify the value. As mentioned earlier, there are limitations that arise based on the condition triggers specified in the type so that the analyses of these triggers can be performed at compile-time. We believe that in many cases these restrictions are reasonable because the modifications made to values are generally done in very controlled manners. In addition, again to allow all analysis to be done at compile-time, modifications to these values must involve constant values which can be evaluated at compile-time to determine whether the modification is allowed and which trigger it activates.

To allow the initialization and destruction of synchronized values, the values can be initialized using `new`, taking an initialized version of the value held within the synchronized type. Synchronized types are also destroyed using `delete`. It is worth noting that this initialization provides a mechanism to circumvent the safety guarantees of the synchronized type, because we could simply reinitialize a value and then multiple threads could conceivably acquire it; this is unfortunately an unavoidable problem, unless we were to utilize significant runtime checks, and simply requires programmers ensure they do not reinitialize synchronized values that are actively in-use.

Finally, we use the conditions by waiting for a condition to be true or to no longer be true. The syntax for this is `wait while` *name*`.`*cond* or `wait until` *name*`.`*cond*, where *name* is the name of a held synchronized value, and *cond* is the name of a condition on that value. A `wait while` means that the threads waits until the condition is signalled to not be true, while `wait until` means that the threads wait until the condition is signalled to be true. There are limitations on these, though; we statically determine if a condition only has signals defined either for the condition being true or only being false, and in that case that condition can only be used with a `wait until` or `wait while` so that we know it could possibly be awoken at some point. After a `wait while`, the expression given with the

condition's definition will be false and after a `wait until` the expression with that condition will be true.

5.2 Implementing a Bounded Buffer

In the previous chapter we provided an explicit definition of a bounded-buffer, it used a lock and two condition variables to achieve the synchronization. Now, consider the semantics of the synchronization for a bounded-buffer. There are two conditions that we are particularly interested in: when the buffer is empty an attempt to remove an element must block and when the buffer is full an attempt to add an element must block. If we have a buffer named `this`, we can express this first condition as `this.n == 0` and the second as `this.n == this.len`. Now, we must also consider what changes made to the buffer cause these conditions to change. We can easily see that when an element is added to the buffer, it is no longer empty and so a thread that was waiting for the buffer not to be empty can be awoken; on the other hand when we remove an element the buffer is clearly no longer full and so a thread that was waiting for the buffer not to be full can be awoken. Thinking about how the conditions can be written as expressions, and what we do when we perform an insertion or deletion, we can determine that when `this.n` is incremented we should signal that the buffer is not empty and when `this.n` is decremented, we should signal that the buffer is not full.

Now, in Figure 5.1 we can see the definition of the synchronized bounded buffer type and the accompanying functions for it. Looking at the synchronized type constructor on line 6, we specify that it holds a `struct bounded_buffer`. We saw line 7 and 8 earlier, and they define the empty and full conditions, matching the semantics we described earlier. We have also seen, lines 10 and 11 before, and they define how to signal these conditions, again matching the semantics we described. The actual implementation of the initialization, get, and put operations are then relatively straightforward. On line 19 we can see the initialization of

the bounded buffer using `new`. We then see the use of `holding` blocks on lines 26 and 38, acquiring atomic access to the bounded buffer before performing the remainder of the get or put operation. We then wait while the buffer is empty, if we are getting a value, or while it is full, if we are adding a value, on lines 27 and 39. From here, we make the needed modifications to the internals of the bounded buffer. We can notice, different from the version presented in the previous chapter, that we make no explicit signals in this code, rather the system will generate the appropriate signals from the actions on lines 31 and 43 which signal, respectively, that the buffer is not full and that it is not empty.

5.3 Silver Implementation

The actual implementation of this extension is somewhat complicated because there is significant difficulty in tracking accesses and actions to determine what actions are allowed and what signals to generate. This is achieved by using the type system in ABLEC and operator overloading; ultimately, a synchronized type has two representations: its synchronized version which, as described above, prevents any access to the contained value; and a held version which is used within a `holding` block as the type of the name of the held value. The first of these types is relatively simple as it simply produces an error message anytime any operation is performed on the value, with the exception of special handling for initialization. This type, though, is also responsible for creating the C implementation of the type. This is actually relatively simple, we produce a `struct` that contains the desired value, a lock of the specified type, and up to two condition variables for each condition one for the condition being true and another for it being false though we do not generate condition variables that are never signalled as waiting on these conditions would never awaken.

Now, the implementation of the `holding` block is also relatively simple: we acquire the lock, create a new variable with the specified name that references the value, execute the body of the block, and then release the lock. In the ABLEC environment, this new name is

```
1  struct bounded_buffer {
2    int head, tail, n, len;
3    struct request** items;
4  };
5
6  typedef blocking synchronized<struct bounded_buffer> = {
7    condition empty(this.n == 0);
8    condition full(this.n == this.len);
9
10   when (this.n) += 1 then signal not empty;
11   when (this.n) -= 1 then signal not full;
12 } bounded_buffer;
13
14 bounded_buffer* create_buffer(int len) {
15   struct bounded_buffer tmp = {0, 0, 0, 0,
16                                malloc(sizeof(struct request*) * len)};
17
18   bounded_buffer* res = malloc(sizeof(bounded_buffer));
19   *res = new bounded_buffer(tmp);
20   return res;
21 }
22
23 struct request* buffer_get(bounded_buffer* buffer) {
24   struct request* res;
25
26   holding (*buffer) as buf {
27     wait while buf.empty;
28
29     res = buf.items[buf.head];
30     buf.head = (buf.head + 1) % buf.len;
31     buf.n -= 1;
32   }
33
34   return res;
35 }
36
37 void buffer_put(bounded_buffer* buffer, struct request* elem) {
38   holding (*buffer) as buf {
39     wait while buf.full;
40
41     buf.items[buf.tail] = elem;
42     buf.tail = (buf.tail + 1) % buf.len;
43     buf.n += 1;
44   }
45 }
```

Figure 5.1: Definition of a bounded buffer using a synchronized type

given a held type, representing that it is a synchronized value within a holding block. In the generate C code, this variable is actually a pointer to the original synchronized value and this pointer is then used to access the value without risk of pointers being changed that break the protections of the synchronized type. The held type is the part actually responsible for determining when signals should be generated, and for checking that the accesses and actions taken to the object are permitted. To do this, the held type can actually represent not only a synchronized value, but also an access into a piece of a synchronized value. For instance, with a bounded-buffer as shown above, `buf` will have a held type representing the entire bounded-buffer but `buf.n` or `buf.len` will also be represented as having a held type. The held type tracks what accesses or actions are allowed on a particular type using a tree-like structure so that it can consider nested accesses to structs or arrays within the actual value's type. The held type uses operator overloading to control what operators are allowed to act on the type, and to then produce the correct held type based on the operator's action. The first restriction, is simply that address of operations are not permitted on a held value, because this could allow access to internal fields that when modified should generate signals. The next case is for operators which produce r-values. An r-value is an expression which is not valid on the left-hand side of an assignment operator, in contrast an l-value is an expression which is valid on the left-hand side of an assignment operator. For instance, `x + 4` is an r-value since we cannot assign to it, while `x.foo` is an l-value since we can assign to it. For operators which produce r-values, the operator is always allowed but it will produce a type which cannot be converted back to an l-value, specifically this prohibits using the produced value with the dereference, arrow, or array access operators since these all convert r-values back to l-values. This restriction is needed to prevent circumventing the signals checks; as an example if we have a synchronized type which contains two pointers `p1` and `p2` and we have this value held as `a` an expression of the following form `*(a.p1 ^ a.p2 ^ a.p1) += 1` (in C, `^` is the exclusive-or operator) can change the value stored at `p2` in a manner that could potentially need to generate a signal, but there is no reasonable way to statically determine

this. Then, for operators that modify the value (such as increment and assignment) we verify that that action is allowed and if so whether there are any signals it must produce, and since the result of these operators is also an r-value the same restrictions as just described are placed on the result of the operand. Finally, with the array access, member, and dereference operators we must determine whether the component we are accessing is associated with any signals and if so produce a held type that enforces these, and otherwise we produce a held type which allows any actions or accesses.

Chapter 6

Performance

We will now look at the performance of code written using some of the parallelization and synchronization constructs we described above. To do this we will consider two separate problems: the first is a compute-bound problem where we are interested in how well the performance scales as more threads are used, the second problem is a blocking-based problem where there are still compute-bound operations but the primary interest is the impact of the locking mechanisms.

These tests were run on a Dell PowerEdge R815 with 4 AMD Opteron 6220 (8-core) CPUs at 3.00 GHz. The machine has 192 GB of RAM. In addition, it was running Ubuntu 18.04.4 LTS and the compilation and linking was performed with GCC version 7.1.0 and GNU `ld` version 2.30, with `-O3` optimization enabled.

6.1 Parallelization

For a compute-bound problem we are using the N-Queens problem, but for simplicity instead of either of the queries described earlier, the problem here is just to count all solutions for a given n. While this obviously has no practical use, since these counts are well known, this problem gives us a highly parallel compute-bound problem and we can easily find values of n to produce sufficiently long test runs.

For this experiment, we compare the performance of a sequential version, and parallel versions using the POSIX, thread pool, and work-stealing systems. Comparing against a sequential version allows us to determine the overhead our systems introduce. We then run each version varying the number of threads from 1 to 32, using the results from having 1 thread as the base-line to determine speed-up and to determine the overhead introduced by the systems. Since our test machine has 32 threads, we do not exceed this number as we expect performance to degrade if we were to.

6.1.1 Test Code

For each system, we define four functions: a setup function which takes as its argument the number of threads that the function is allowed to utilize, a tear-down functions, a helper function used for the searching, and the actual function that is invoked to count the solutions for a given n. The code is relatively similar in each case, and generally has the form shown in Figure 6.1. In this code, `some` represents any of the parallel system qualifiers that we have described.

Next, we consider the differences between the implementations for each of the different systems. The POSIX and thread pool implementations are very similar. For the POSIX and thread pool implementations the `count_arrangements` function is shown in Figure 6.2. This code uses a parallel for-loop to divide the counting into pieces that are performed in parallel. Now, because the machine has 32 threads and we will be running these with n no larger than 16, the code for POSIX and thread pools actually parallelize a loop that fills the first two rows of the board so that there are sufficiently many pieces to divide amongst 32 threads. Because this produces a number of iterations significantly larger than the number of threads, we expect reasonably good linear speedups from this code. The code here uses an atomic integer as the counter, we believe this to be reasonable since there are still relatively few times the counter will be modified and so there should be relatively little contention. Finally, the one major difference in the implementation between POSIX and thread pool is

```
some parallel threads;
int nThreads;

void setup(int t) {
  threads = new some parallel(t);
  nThreads = t;
}

void teardown() {
  delete threads;
}

int count_helper(chess_board* board, int n, int r) {
  if (r == n) return 1;

  int cnt = 0;
  for (int c = 0; c < n; c++) {
    if (nqueens_filled(board, r, c)) continue;

    nqueens_set(board, r, c);
    cnt += count_helper(board, n, r+1);
    nqueens_unset(board, r, c);
  }

  return cnt;
}
```

Figure 6.1: Outline of the N-Queens counting code

```
int count_arrangements(int n) {
  if (n == 1) return 1;

  posix group grp; grp = new posix group();

  _Atomic int count = 0;
  parallel for (int i = 0; i < n*n; i++) { by threads; in grp;
                        num_threads ts; private n; global count_helper;
                        public count; global setup_board;
    int c0 = i / n, c1 = i % n;
    chess_board* board = setup_board(n, c0, c1);
    if (board) {
      count += count_helper(board, n, 2);
    }
  }

  sync grp; delete grp;

  return count;
}
```

Figure 6.2: Outline of the actual counting function for POSIX and threadpool

```
1  int count_arrangements(int n) {
2    posix group grp; grp = new posix group();
3
4    chess_board* board = create_board(n);
5
6    int count;
7    spawn count = count_helper(board, n, 0); by threads; in grp;
8
9    sync gry; delete grp;
10
11   return count;
12 }
```

Figure 6.3: Work-stealing code for count arrangements

the value of `num_threads`; in the POSIX implementation this must be the number of threads we want to produce while for the thread pool version we let it be n^2 because that allows each iteration to be executed separately which should allow the thread pool to operate as a load-balancer.

The work-stealing implementation is very different; the code for the helper function is the same as shown in Figure 3.3, and the major difference from the other systems is that the recursive calls to the helper function are spawned so that they can be performed in parallel. Because the helper function introduces parallelism, the `count_arrangements` function does not need to parallelize anything, and so it simply creates an empty board and spawns the helper function to run. The code for this is shown in Figure 6.3.

6.1.2 Results

To measure the times, we used the C standard library function `clock_gettime`. It was used to measure ten iterations of `count_arrangements` to help account for differences between executions. As stated, we ran each test starting with 1 thread and then every value up to 32 threads. We used n values of 14, 15, and 16 because we found that these required reasonable execution times, long enough that we believe the overhead of the timing functions should be negligible and short enough to complete the tests in a reasonable length of time. The full

results are available online[1]. For $n = 16$ the speed-up compared for each system individually to their performance with one thread is shown in Figure 6.4. These results are similar to those for $n = 14$ and 15. With $n = 16$ we measured about 24 times speed-up with the thread pool, 20 times speed-up with the POSIX system, and 16 times speed-up with the work-stealing system, when running with 32 threads.

The remaining component that is of interest is the overheads each system imposed. For the POSIX and thread pool systems, the overhead was very consistent for all three values of n and was about a 5% slow-down compared to the sequential version. For the work-stealer, the overhead was less consistent: for $n = 14$ the overhead was around 775%, for $n = 15$ it was around 755%, and for $n = 16$ it was around 735%. The very large overhead from the work-stealing system is not completely unexpected as Cilk5 has significant overhead as well, though significantly smaller than these. Some profiling of work-stealing code has suggested that the significant overhead is a result of memory allocation relating to the closures, as well as contention on the locks used to synchronized the deques.

Looking now to Figure 6.4 we can observe the speed-ups produced by each system compared with the number of threads. In this, each system's speedup is computed from the observed latency of that system when using only one thread. Looking to Figure 6.4 we can see that all three systems seem to provide reasonably good linear speed-up, though the thread pool seems to perform significantly better than the POSIX or work-stealer systems. The very good performance of the thread pool is likely due to its ability to act as a load balancer. The performance of the work-stealer is actually somewhat underwhelming since the main benefit of work-stealing is that it is expected to work very well as a load-balancer. Further analysis is needed, but it is possible that the memory allocation and lock contention problems which cause it to have high overhead are also causing a bottle-neck that is limiting performance, especially since we utilize the `malloc` and `free` implementations provided in glibc which make some use of mutexes, which could be limiting parallel performance.

[1] https://github.com/melt-umn/ableC-parallel/blob/e70fb2562b42bfba7c19aef248f1fb2bf45077ff/examples/results.csv

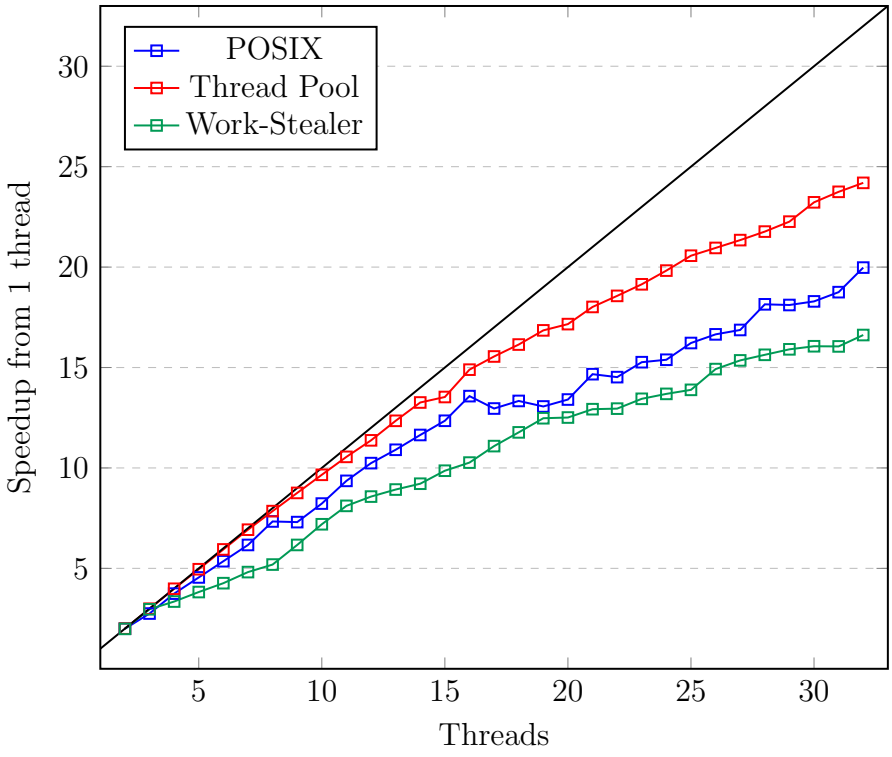

Figure 6.4: Speed-up of counting N-Queens Solutions for $N = 16$

6.2 Synchronization

For our synchronization performance tests, we are interested in comparing the performance of the POSIX and blocking implementations. Because the POSIX implementation translates directly into use of POSIX mutexes and condition variables, with a little bit of additional error checking, we expect the performance difference between the POSIX system and manual use of POSIX mutexes and condition variables to be minimal. Therefore, what we are really interested in is testing the blocking implementation to see if it can ever outperform the POSIX implementation. The situation we expect this to occur is when using a thread pool that has more work items than OS threads and when there is sufficient contention that threads block for significant lengths of time during which progress could be made on other work items.

The problem that we developed for this test is, unfortunately, rather contrived, but

demonstrates the performance and potential of these two different locking systems. The problem involves a set of n prime numbers and thirty threads, each of which make a hundred thousand random *accesses* to these numbers. When a prime is accessed that prime becomes invalid until its value is updated to be that of the next larger prime. This process, of finding large primes, is where the significant portions of compute-bound work are performed in this problem, especially since the implementation used for finding primes is not optimal. In full transparency, the number of threads used was chosen to generate contention.

The parallelism in this problem comes from the 30 threads and, in addition, the task of finding the next prime and update the value is also spawned as a new task. Both of these types of threads are created using the same parallelization technique, so for this test either using the POSIX system or the same thread pool with thirty-two threads.

The synchronization in this problem comes from our representation of the prime numbers, which is shown in Figure 6.5. Here, we use a synchronized type, and actually demonstrate some techniques not shown previously. To access a prime, we then gain exclusive access to the prime using `holding`, wait for the prime to become "valid", read the value out of it, and then mark the prime as invalid again. We then spawn a task to update the prime and return. Updating the prime is simply a process of finding the next prime.

Part of what makes this a useful example is that for sufficiently large n we may have many update tasks running in parallel, and since they are compute-bound we expect degradation in performance if we use OS threads. On the other hand, with smaller values of n we expect the performance to be largely based on the lock contention, since there are fewer update tasks running in parallel.

6.2.1 Results

To collect these results we ran the test described above ten times, resetting the primes in between and then averaging the running times of each of the ten repetitions. The program was run under a total of 16 configurations, we had n values of 8, 16, 32, and 64; the

```
1  typedef struct { unsigned long val; short int ready; } pair;
2
3  typedef some synchronized<pair> = {
4    condition valid(this.ready);
5
6    when (this.ready) == 1 then signal valid;
7    when (this.ready) == 0 then ignore;
8  } prime;
9
10 void update_prime(prime* prime) {
11   holding (*prime) as p {
12     p.val = next_prime((unsigned long)(p.val + 1));
13     p.ready = 1;
14   }
15 }
16
17 unsigned long get_prime(prime* prime) {
18   unsigned long res;
19   holding (*prime) as p {
20     wait until p.valid;
21     res = p.val;
22     p.ready = 0;
23   }
24
25   spawn update_prime(prime);
26           by thrds; in grp; private prime; global update_prime;
27   return res;
28 }
```

Figure 6.5: Code for working with the "primes" used by the synchronization problem

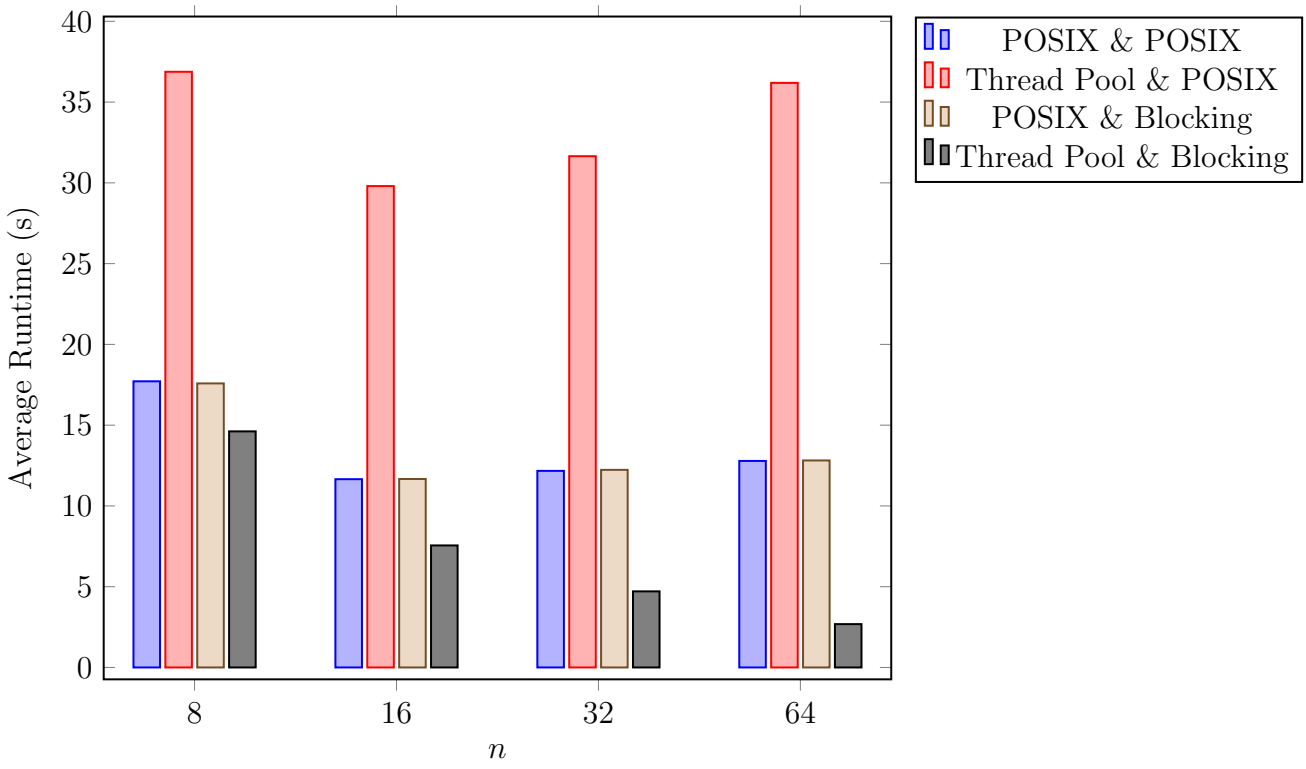

Figure 6.6: Synchronization Test Results, labels in the form "parallel system & synchronization system"

parallelism was achieved using the POSIX and thread pool systems; and the synchronization was achieved using the POSIX and blocking implementations. The full results are shown in Table A.1 of Appendix A. These results are also plotted in Figure 6.6.

Immediately, we see that using a thread pool with POSIX synchronization has pretty bad performance which is expected since most likely thirty of the threads are accessing values, and only two or so remain to actually handle updating the primes. We can also note that the thread pool using blocking synchronization is very fast, especially for high n values. This is to be expected since the higher values of n should have lower contention while having more update tasks running in parallel, the thread pool performs better at this than the POSIX system likely because it ensures that the total number of running threads never exceeds the number of cores on the machine. We see the gap narrowing as n decreases because the number of update tasks running in parallel decreases.

We can also see that the performance using POSIX parallelization and comparing POSIX versus blocking synchronization is nearly identical. From the data we actually find that with $n = 8$ the blocking locks are less than 0.5% faster, for $n = 16$ the POSIX locks are after by less than 0.1%, for $n = 32$ they are about 0.5% faster, and for $n = 64$ they are about 0.2% faster. This suggests that the overhead of the blocking locks compared to POSIX locks is, surprisingly, minimal and with high enough contention the blocking implementation may actually be faster.

All together, this suggests that the blocking lock system has reasonable performance compared to the POSIX lock system. Furthermore, it shows that in certain situations, when these locks are needed when working with a thread pool, they can also have major performance improvements. Finally, it shows that using a thread pool to avoid having too many OS threads running compute-bound tasks can have major performance benefits as well.

Chapter 7

Related Work

There have been many previous attempts to develop languages well suited to parallel programming, and many such languages exist, though this remains an open problem in language design. Previous attempts at developing parallel programming features generally fall into one of three categories: libraries, extensions to existing languages, and new programming languages. Each of these techniques, and individual implementations, has its unique advantages and disadvantages; we will discuss several of these below.

7.1 Parallel Libraries

As we have already seen, parallel programming in C itself is done through the `pthread` library. As we have mentioned already, this approach requires significant amounts of boilerplate code. In addition, because a library does not introduce any new syntax to the language, invoking the library is just done through function calls, which makes the code less easy to understand, and because the `pthread` implementation relies on void pointers, there is minimal type-checking enforced on these operations.

Parallelism is also introduced via libraries in Java; threads an be created directly using the `java.lang.Thread` class as well as using the `java.util.concurrent.ExecutorService` interface, and the implementations of it provided in that same library. Considering the

`ExecutorService`, there are three major differences between how parallelism is achieved in Java and in C, despite both using libraries. First, generics in Java avoid the type-checking problems that arise from the use of void pointers in C. Secondly, since the introduction of lambda expressions (anonymous functions) in Java 8, task spawning can be written using a more natural syntax which avoids the boiler-plate code of either an extra class or an anonymous class. Finally, Java's use of interfaces with polymorphism allows the implementation of the `ExecutorService` to easily be changed, even possibly at runtime, to any class which implements that interface.

Another example, though one that borders on a language extension, is Halide[14], which is an embedded domain-specific language (DSL) in C++. While Halide appears to be a DSL, it actually makes use of operator overloading in C++ rather than actually introducing new syntax, and so is actually a library. Halide is designed for writing efficient image processing code; it allows programmers to separately specify the semantics of loops and various techniques that can be used to optimize them, including parallelization of these loops. Halide was a strong inspiration of this work: the idea of separating the semantics of a program from its implementation, and specifying implementation details using annotations inspired the approach to parallelism that we took in this work.

These examples present very different views of how a parallel programming library can function, and there are some shared benefits: libraries are easy to introduce since they can generally be written in the source language and as such do not require the design of any translation or type-checking systems. In addition, programmers can easily choose which libraries they want to use, rather than picking a language which has significant impact on not only design but also the build process. This ability to pick the libraries that a programmer needs, is also a benefit of extensible programming languages, where the programmer can pick the language extensions that fit their project. On the other hand, libraries cannot introduce new static analysis or type checking, beyond that provided in the language, which is a significant disadvantage since, for example, the type-based synchronization discussed in

Chapter 5 is impossible as a library, at least without significant runtime checks.

7.2 Parallel Language Extensions

There have also been many attempts to add parallel programming features to existing languages, the general idea being that it is easier to get programmers to use an extended version of the language, rather than convincing them to learn an entirely new language. We have already discussed OpenMP and Cilk which are both examples of language extensions, OpenMP is an extension to C/C++ and Fortran, and Cilk to C/C++. At least in C, OpenMP and Cilk both introduce parallelism through relatively minor changes to sequential C code: for OpenMP we add `#pragma` annotations which can be added directly to normal C code, and `cilk`, `spawn`, and `sync` keywords which can be added to regular C code to produce parallel Cilk code.

Another example is the X10[5] language, developed as part of DARPA's "High Productivity Computing Systems" project, which was a major project started in 2002 to develop parallel programming languages that were easier to use than existed at the time[18]. X10 was developed as an extension to Java with additional features useful for parallel programming[12]. X10 is actually a decent example of the limitations of parallel programming extensions: because it kept everything present in the Java language, X10 ended up with separate definitions of classes whose objects are passed by reference (like objects in Java) and classes whose objects are passed by value (like classes in C++). While this may not inherently be a bad design, it does demonstrate some of the potentially odd designs that are forced by extending another language.

Another example particularly interesting to our work is Delite[3]. Delite is a framework and runtime system designed to aid in creating domain-specific languages for parallel programming, specifically implicitly parallel DSLs. We mention Delite because extensible programming languages like ableC can be thought of as a tool for building programming

languages, in ableC we build extensions that generally introduce new features, rather than building entire new languages. There are also similarities in that ableC parallel itself operates as a framework for building higher-level parallel programming extensions, or it could even be used as the target language for a parallel programming language.

Finally, this work is based on ableC which is an extensible C preprocessor. Using ableC, it is very easy to develop language extensions as we do not have to write a new compiler or modify an existing one, rather we develop and extension which is also composable with other extensions. There are of course, limitations to this technique since an extension cannot remove features or syntax from the base language.

Ideally, there are two advantages of language extensions for parallel programming over a new language: first, programmers are already familiar with most of the language, so they are more likely to adopt it; and second, it can be easier to develop, especially if an existing compiler can be extended to support the new language features. However, this is not necessarily the case. For instance, the original translator from Cilk5 to C code included its own parser and type checker.

7.3 Parallel Programming Languages

Our first two examples come from the aforementioned DARPA project. These languages are Fortress[16] and Chapel[4]. Fortress was designed almost to be an extensible language, it supports operator overloading but also allows the introduction of new operators, which are allowed to use characters from the entire Unicode standard. Fortress achieves parallelism "by-default", for instance every for-loop is parallel unless it is marked as sequential. A more modern example of a programming language designed for parallel programming is Go[7]. Go provides a variety of primitives for creating threads and passing messages between them to achieve synchronization.

There are many examples of parallel programming languages, we will mention several

others that are particularly interesting to us, especially from a programming languages perspective. One of these is Single Assignment C (SaC)[15]. This language is based largely on the syntax of C, but limits what is permitted so as to make the language a functional programming language without side-effects but still with support for $O(1)$ multi-dimensional array accesses. The reasoning for this is that a functional programming language without side-effects can be much more easily analyzed and implicitly parallelized than is possible in an imperative language which is based on side-effects. One of the design ideas behind SaC is that because it has syntax similar to that of C, there are programmers who might otherwise be intimidated to program in a functional language but might be more comfortable working in SaC.

Many of the languages we have discussed so far have had relatively limited adoption, and much of this is a result of non-technical aspects related to language preferences and the difficulty of changing languages. An exception to this, is Go, which is becoming a rather popular language at this time[2]. The diversity of work in this area demonstrates the difficulty of designing good parallel programming languages. Because C is a relatively common language, we do believe that our work could be relatively adoptable and we believe that the extensible nature is also useful as it gives programmers more flexibility in picking the features they want to use, or even developing new ones.

Chapter 8

Future Work

We will now discuss a variety of work that remains to be done, or interesting work that we believe could be added to this system to make it more useful. We will discuss, first, a number of improvements to the current system and extensions and then we mention several other extensions we believe would be interesting to develop.

8.1 Improvements

We will first consider several ways we believe this work can be improved. The first of these involves changes to the annotations used for parallelism, we also consider ways that the performance of the system could be improved, and finally discuss some performance improvements that could be made to the work-stealing system as well as addressing some of the incomplete aspects.

8.1.1 Annotations

As mention in Chapter 3 several of the annotations used for specifying parallelism have semantic meaning, even though the original intention was for them to have no semantic meaning. This is a flaw because it blurs the line between the semantic portions of the program

that should be able to be reasoned about in a sensible manner and the implementation details which should be used in tuning performance but should not impact the semantics of the program at all. Even though each annotation is either semantic or implementation, the fact that they are represented by similar syntax makes it difficult to distinguish between them, and so requires reading all the annotations to understand the semantics of the program. Further work on this system should consider other techniques of representing these semantic annotations that would allow for a clean distinction between semantic and implementation details.

In addition, this system is designed to be as extensible as possible, however the annotations themselves are not. The annotations are specified in the base system and each extension that provides a parallel implementation must handle each annotation. This means it is difficult to add new annotations, and so future work should attempt to develop annotations in such a manner that they can be developed as separate extensions and plugged into different parallel implementations.

Finally, there are also a lot of annotations required in the system's current form because there are no default behaviors at all. A good example of this is the sharing annotations, where reasonable choices between public, private, and global can likely be inferred by analysis. Another example might be the `num_threads` annotation on parallel for-loops where we might actually want it to default to either the number of threads in a thread pool or to the number of threads supported by the hardware. Future work could improve usability of the system by developing sensible default choices for various annotations to reduce the number of annotations that programmers need to write.

8.1.2 Performance

Many of the features included in the system could likely benefit from further performance analysis and re-writing to improve performance. This is especially the case for the thread pool implementation; currently, the system uses a single work queue synchronized using

POSIX locks, so there may be better implementations or better synchronization techniques since in some cases the queue may become a point of high contention. In addition, currently the thread pool acquires and releases the lock repeatedly when adding work items for a parallel for-loop while we could instead build up a list of the items and then add them all at once to reduce contention.

Additionally, the current implementation of blocking locks could perhaps be improved borrowing techniques from Linux "futex" locks by spinning for a little before actually blocking, since blocking and unblocking are relatively expensive and in some settings locks are not held for very long.

8.1.3 Work-Stealer

There are a variety of ways that the work-stealing system can be improved, and some of these have been outlined above. The first, as shown in Chapter 6 the system has quite a large overhead. Some profiling has suggested that this overhead is in large part due to memory allocation and the synchronization used for the deques. The first of these issues could potentially be resolved using a thread-cached memory allocator and the later of these could be perhaps be improved by using a more efficient deque implementation; the current implementation relies on a single POSIX lock while the original Cilk5 paper[8] describes a generally lock-free implementation that might help reduce the overhead of the system.

There are also a variety of details that are not completed by the work-stealing system, specifically it lacks support for parallel for-loops and does not allocate TCBs for each thread or work with the blocking synchronization extension. The solutions to these first two are relatively simple, but allowing support for blocking will be more difficult. It likely involves a technique similar to how blocking is achieved in the thread pool system, perhaps by having each work thread have two stacks: one used in the scheduler and another used when performing work, and when we need to block jumping back into the scheduler and then allocating a new work stack; resumption of a blocked task would then involve jumping back

to the original location in code and on the original stack.

8.2 Further Extensions

Finally, we believe there is significant room for further extensions to be developed, either to provide new abstractions or to provide new implementations. Some examples that we have considered would be to introduce message passing abstractions, implementation-agnostic atomic types that could then be implemented using hardware synchronization or various lock implementations, and other synchronization techniques like barriers or read-write locks.

Bibliography

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992. doi:10.1145/146941.146944.

[2] Pierre Carbonnelle. PYPL popularity of programming language index, 2020. URL https://pypl.github.io/PYPL.html. Accessed Apr. 14, 2021.

[3] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, page 35–46, Feb 2011. doi:10.1145/2038037.1941561.

[4] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the Chapel language. *International Journal of High Performance Computing Applications*, 21(3), 2017. doi:10.1177/1094342007078442.

[5] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 519–536, October 2005. doi:10.1145/1103845.1094852.

[6] Aaron Councilman. Extensible parallel programming in ABLEC. https://hdl.handle.net/11299/203190, May 2019.

[7] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[8] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings on the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM. doi:10.1145/277650.277725.

[9] Ian P. Gent, Christopher Jefferson, and Peter Nightingale. Complexity of n-queens completion. *Journal of Artifician Intelligence Research*, pages 815–848, August 2017. doi:10.1613/jair.5512.

[10] *POSIX.1-2017*. IEEE, 2017.

[11] Ted Kaminski, Lucas Kramer, Travis Carlson, and Eric Van Wyk. Reliable and automatic composition of language extensions to C: The ableC extensible language framework. *Proceedings of the ACM on Programming Languages*, OOPSLA(1), October 2017. doi:10.1145/3138224.

[12] Ewing Lusk and Katherine Yelick. Languages for high-productivity computing: The DARPA HPCS language project. *Parallel Processing Letters*, 17(1), January 2007. doi:10.1142/S0129626407002892.

[13] *OpenMP Application Programming Interface*. OpenMP Architecture Review Board, November 2018.

[14] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Transactions on Graphics*, 31(4), July 2012. doi:10.1145/2185520.2185528.

[15] Sven-Bodo Scholz. Single Assignment C-efficient support for high-level array operations in a functional setting. *Journal of functional programming*, 13(6), November 2003. doi:10.1017/S0956796802004458.

[16] Guy Steele. Parallel programming and parallel abstractions in Fortress. https://www.brics.dk/pilambda/old/docs/Aarhus-Fortress-Parallelism-2006public.pdf, October 2016.

[17] Ruud van der Pas. OpenMP tasking explained. https://www.openmp.org//wp-content/uploads/sc13.tasking.ruud.pdf, November 2013.

[18] Michèle Weiland. Chapel, Fortress and X10: novel languages for HPC. Technical Report 0706, EPCC, The University of Edinburgh, 2007.

Appendix A

Full Performance Results

n	Parallelism	Synchronization	Runtime (s)
8	posix	posix	17.7149457129
8	thrdpool	posix	36.8770649055
8	posix	blocking	17.5840930883
8	thrdpool	blocking	14.6150645092
16	posix	posix	11.6559275858
16	thrdpool	posix	29.8012408274
16	posix	blocking	11.6667244879
16	thrdpool	blocking	7.5544029518
32	posix	posix	12.1698284111
32	thrdpool	posix	31.65075647
32	posix	blocking	12.2327220253
32	thrdpool	blocking	4.7112120074
64	posix	posix	12.7849644252
64	thrdpool	posix	36.190214075
64	posix	blocking	12.8149937422
64	thrdpool	blocking	2.6833749939

Table A.1: Full Results of Synchronization Tests